EXPERIMENTAL RESULTS ON SOFTWARE DEBUGGING

Sylvia B. Sheppard
Phil Milliman
Bill Curtis
Software Management Research
Information Systems Programs
General Electric Company
1755 Jefferson Davis Highway
Arlington, VA 22202

INTRODUCTION

Debugging programs is one of the most expensive, time-consuming activities in the development of a software system. Only a few laboratory experiments have investigated the relative difficulty of locating different types of bugs or the most effective search strategies. Youngs (1974) found that experience contributed to differences among types of errors made in a construction experiment. Wescourt and Hemphill (1978) described a model of the debugging process, but the model was not entirely supported by the available data. Gould and his associates (Gould and Drongowski, 1974; Gould, 1975) found that the type of bug influenced debugging performance on short programs. Specifically, assignment bugs were more difficult to locate than array or iteration bugs, probably because the former required a greater understanding of the algorithm used by the prograr

The difficulty of debugging a program may be associated with coding practices used during its development. One factor which may influence the ease of finding a bug is the complexity of a program's control flow. Two previous experiments by the authors investigated the effects of structured control flow in understanding and modification tasks (Sheppard, Curtis, Borst, Milliman, and Love, 1979). Programmers performed their tasks more efficiently on code which exhibited a straightforward, top-down control flow than on an unstructured, convoluted control flow. A rigorously structured control flow (Dijkstra, 1972) did not produce significantly better performance than a naturally structured version which allowed limited unstructured constructs (e.g., exits from loops). Thus the overall top-down quality of the control flow appears to influence performance, while minor deviations from the tenets of structured code do not appear to influence performance significantly. This result may reflect the innate awkwardness of implementing strictly structured code in standard Fortran.

Factors other than the structuredness of the control flow may influence the complexity of a computer program and, thus, the difficulty programmers experience in performing their tasks. Some of these factors have been quantified in the software complexity metrics developed by Halstead (1977) and McCabe (1976). Halstead's metric purportedly represents the number of mental discriminations involved in developing a program, while McCabe's metric measures the number of elementary control path segments comprising a program. In experiments on understanding and modification, these software complexity metrics were evaluated for their usefulness as predictors of programmer performance (Curtis, Sheppard, Milliman, Borst, and Love, 1979). The results observed in those experiments were modest. The correlations in the raw data were not large, and the number of lines of code usually predicted programmer performance better than the Halstead or McCabe metrics. Several limitations in the experimental procedures employed to obtain the data may have produced these results. First all of the programs studied were short

(35–55 lines of code). The limited range of metric values calculated on programs of this length may not have been sufficient for an adequate test of the predictive worth of the metrics. Second, individual differences among programmers exerted significant effects on the results obtained. When the data from the first experiment were transformed in an attempt to control for differences among programs and programmers, a correlation of -0.73 ($p < 0.001$) was obtained between the performance criterion and Halstead's E. However, the issue is not whether theories can be validated with mystical transformations of data, but whether the results of these heuristic transformations can be replicated in an experiment designed to overcome the limitations of previous research.

The present experiment evaluated the difficulty of locating three types of errors under controlled programming conditions. In order to compare the effects on performance of different methods of structuring code, programs in the present experiment were implemented in three types of control flow, all of which exhibited a generally top–down flow. This experiment also evaluated the ability of software complexity metrics to predict performance over a wider range of program sizes. To investigate the effects of length, the three programs in this experiment were subdivided into functional subroutines so that they could be presented in three different lengths: approximately 50, 125, and 200 lines of code. Finally, the present experiment attempted to relate programming performance to experiential factors, such as familiarity with other programming languages or relevant programming tools and concepts.

METHOD

Participants

Fifty–four professional programmers at six different locations participated in this experiment. Thirty were civilian employees, while 24 were employees of the military. The participants averaged 6.6 years of professional experience programming in Fortran, ranging from 1/2 year to 25 years (SD = 6.1).

Experimental Design

In order to control for individual differences in performance, a within–subjects, 3^4 factorial design was employed. Three types of control flow were defined for each of three programs, and each of these nine versions was presented in three lengths with three different bugs, for a total of 81 different experimental conditions. The first 27 participants each saw three of the programs, exhausting the 81 conditions (Fig. 1). The second set of 27 participants replicated the conditions exactly except that the order of presentation of the tasks was different in each case.

Learning effects were expected on the basis of results obtained in previous experiments of this type (Sheppard, Curtis, Borst, Milliman, and Love, 1979; Sheppard and Love, 1977). Therefore, the order of presentation of conditions was counterbalanced to assure that each level of each independent variable appeared as the first, second, or third task an equal number of times.

Procedure

A packet of materials prepared for each participant included: (1) written instructions on the experimental tasks, (2) a short tutorial of commands used in Fortran 77, (3) a short preliminary task (Appendix A), (4) three experimental tasks, and (5) a questionnaire concerning previous experience.

PROGRAM	LENGTH	NATURALLY STRUCTURED			GRAPH-STRUCTURED			FORTRAN 77			← CONTROL FLOW
		1	2	3	1	2	3	1	2	3	← BUG
1 ROOTS	SHORT	1	23	12	20	15	3	18	2	26	
	MEDIUM	19	11	7	14	9	25	8	22	17	
	LONG	10	4	27	6	24	15	21	16	5	
2 ACCT	SHORT	13	8	21	7	27	16	24	10	9	
	MEDIUM	5	26	15	23	18	4	12	6	20	
	LONG	22	14	2	17	1	19	3	25	11	
3 GRADER	SHORT	25	17	6	11	5	22	4	19	14	
	MEDIUM	16	3	24	2	21	10	27	13	1	
	LONG	9	20	18	26	12	8	15	7	23	

EACH CELL REPRESENTS ONE OF THE THREE TASKS GIVEN TO A PARTICIPANT

Figure 1. Assignments of 27 Participants in One Replication of the Experimental Design

All tasks included input files, a listing of the Fortran program with the embedded bug, a correct output, and the erroneous output produced by this program. All differences between the correct and erroneous output were circled on the erroneous output. Also included were explanatory descriptions of any subroutines or functions not presented in the listing but referenced by the program.

The 54 participants were divided into two groups of 27, each of which represented a complete replication of the design. Within a group all participants were given the same preliminary task. Group 1 worked with an algorithm to find the greatest common divisor of two numbers and Group 2 was given a simple sort algorithm. These preliminary tasks were provided to reduce learning effects on the experimental tasks and to provide a basis for comparing the abilities of the participants to perform a task of this nature.

Following the initial exercises, participants were presented with three separate programs comprising their experimental tasks. Participants were allowed to work at their own pace, signalling the experimenter when they believed they had identified and corrected the bug. The experimenter verified all corrections, and in the case of a mistake the participant was instructed to try again until the task was successfully completed. The maximum time participants were allowed to work on a particular program was 45 minutes for the preliminary task and 60 minutes for each experimental task. Time was measured to the nearest minute.

Independent Variables

Program. Three programs were selected for the generality of their content and their understandability to programmers. The first program sorted and categorized alphabetic response data to a questionnaire (Veldman, 1967). The second program, an accounting routine, produced income and balance statements (Nolen, 1971). Program 3 kept track of students' test grades and calculated their semester averages (Brooks, 1978). All programs were tested prior to the experiment.

Length. The inclusion of additional subroutines made it possible to present each program in three different lengths. The shorter programs had 25–75 statements, medium programs contained 100–150 statements, and the longer programs contained approximately 175–225 statements. (One Fortran 77 version exceeded the 225 line limit by 8 lines because of the number of ELSE and ENDIF statements required.)

Program listings included a two or three line explanation of any routine or function that was called by a program but not presented in the experimental materials. Participants were told to assume that missing routines worked correctly. All of the input and output files were presented regardless of the length of the program. That is, for the shorter version, some of the input was read in and some of the output was produced by subroutines which were not presented.

Complexity of Control Flow. Three versions of control flow performing identical tasks were defined for each program. Two types of structures were implemented in Fortran IV, naturally structured and graph-structured. A third version was written in Fortran 77 (Brainerd, 1978), which includes the IF-THEN-ELSE, DO-WHILE, and DO-UNTIL constructs.

The Fortran 77 version of each program was implemented in a precisely structured manner. All flow proceeded from top to bottom, and only three basis control constructs were allowed: the linear sequence, structured selection, and structured iteration (Fig. 2).

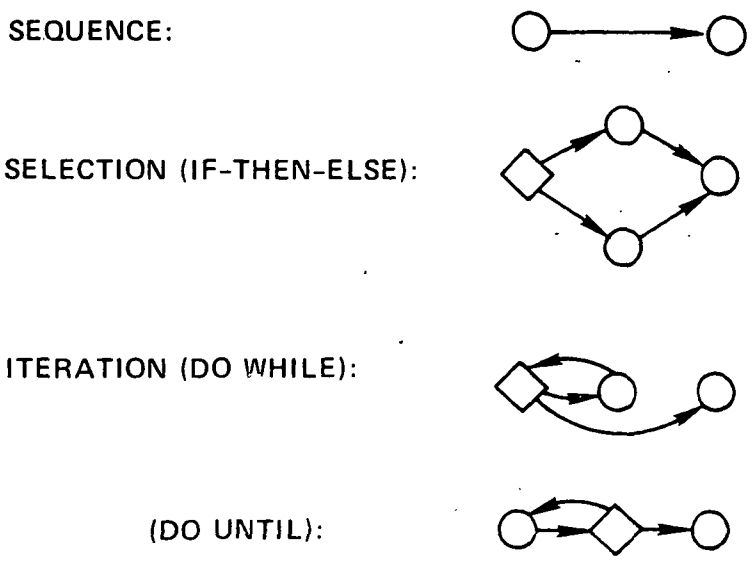


Figure 2. The Basic Structured Constructs

The graph-structured version of each program was implemented in Fortran IV from the Fortran 77 version, replacing the special constructs but producing code for which the control flow graphs of the two versions were identical. All nested relationships could be reduced through structured decomposition to a linear sequence of unit complexity. A full discussion of reducibility is presented by McCabe (1976).

Structured constructs are awkward to implement in Fortran IV (Tenny, 1974). In order to test a more naturally structured flow, limited deviations were allowed in a third version of each program. These deviations included such practices as branching into or out of a loop or decision and multiple returns. Control flow graphs and the code for a section of a routine implemented in all three versions of control flow are presented in Figures 3 and 4.

Each program was indented following the nesting patterns presented in the code. Thus, all DO loops and branching instructions were indented. For naturally structured versions, decisions were made arbitrarily about the importance of various constructions, and indenting was necessarily less standardized than for the graph-structured and Fortran 77 versions.

Type of Bug. Three types of semantic bugs were chosen from a classification developed by Hecht, Sturm, and Trattner (1978): computational, logical, and data errors. Bugs in each category were defined for each of the three programs in order to maximize the similiarity of bugs from a single category across programs. Computational bugs involved a sign change in an arithmetic expression. Logic bugs were implemented by using the wrong logical operator in an IF condition. Data bugs involved wrong index values for variables.

Each bug in this experiment was purposely designed to affect only a limited area of code. That is, each calculation containing a bug occurred near the corresponding WRITE and FORMAT statements. In no case did a bug produce errors in routines other than the one in which it was embedded, and each bug appeared in only one line of code.

Individual Differences Measures

Scores on the preliminary exercise were used as a measure of programming ability related to the experimental task. Participants were also asked to complete a questionnaire about their programming experience. The information required included specific types of experience, number of years programming professionally in Fortran, number of statements in the longest Fortran and non-Fortran programs written, the first programming language learned, and number of languages learned. In addition, various programming concepts that appeared relevant to the experimental programs were listed, and participants were asked to mark those with which they were familiar.

Complexity Metrics

Halstead's E. Using a program based on Ottenstein (1976), Halstead's effort metric (E) was computed from the source code listings of the 27 experimental programs, representing three distinct programs at three levels of structure and three different lengths. The computational formula was:

$$E = \frac{\eta_1 N_2 (N_1 + N_2) \log_2 (\eta_1 + \eta_2)}{2\eta_2}$$

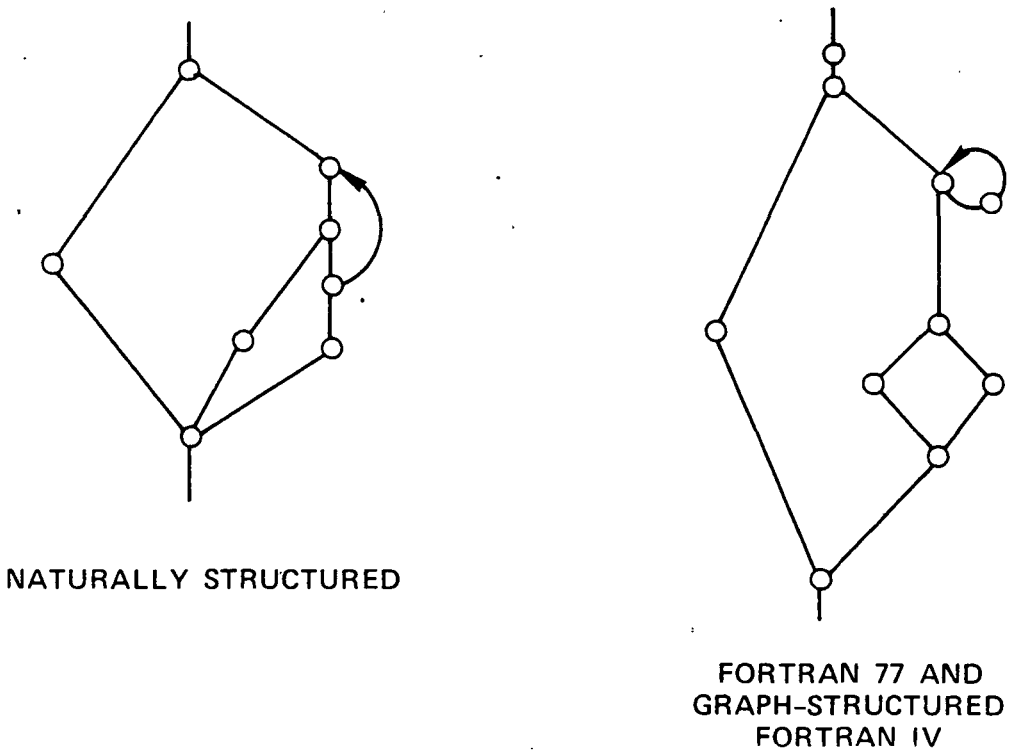


Figure 3. Control Graphs for All Versions of Control Flow

NATURALLY STRUCTURED

```
      IF (ASNUM .LT. 1 .OR. ASNUM .GT. NASSGN) GO TO 420
         DO 400 K=1,NSTUDN
            IF (CURID .EQ. ID(K)) GO TO 440
400         CONTINUE
         PRINT 410,CURID
410      FORMAT (1H0,30X,' ID NUMBER NOT IN FILE:  ',I8)
         GO TO 450
420      PRINT 430, CURID,ASNUM
430      FORMAT (1H0,30X,'ID ',I8,'ILLEGAL ASSIGNMENT',I3)
         GO TO 450
440   SCORE(K,ASNUM)=VAL
450   CONTINUE
```

GRAPH-STRUCTURED

```
      K=1
      IF (ASNUM .LT. 1 .OR. ASNUM .GT. NASSGN) GO TO 420
400      IF (CURID .EQ. ID(K) .OR. K .GT. NSTUDN) GO TO 405
            K=K+1
            GO TO 400
405      IF (K .LE. NSTUDN) GO TO 415
            PRINT 410,CURID
410         FORMAT (1H0,30X,' ID NUMBER NOT IN FILE:  ',I8)
            GO TO 450
415         SCORE(K,ASNUM)=VAL
            GO TO 450
420      PRINT 430, CURID,ASNUM
430      FORMAT (1H0,30X,'ID ',I8,' ILLEGAL ASSIGNMENT ',I3)
450      CONTINUE
```

FORTRAN 77

```
      K=1
      IF (ASNUM .GE. 1 .AND. ASNUM .LE. NASSGN) THEN
            DO 400 WHILE (CURID .NE. ID(K) .AND. K .LE. NSTUDN)
400            K=K+1
               IF (K .GT. NSTUDN) THEN
                  PRINT 410,CURID
410               FORMAT (1H0,30X,' ID NUMBER NOT IN FILE:  ',I8)
               ELSE
                  SCORE(K,ASNUM)=VAL
               ENDIF
         ELSE
            PRINT 430, CURID,ASNUM
430         FORMAT (1H0,30X,'ID ',I8,' ILLEGAL ASSIGNMENT ',I3)
         ENDIF
450   CONTINUE
```

Figure 4. Examples of the Three Types of Control Flow

where,

η_1 = number of unique operators

η_2 = number of unique operands

N_1 = total frequency of operators

N_2 = total frequency of operands

McCabe's v(G). McCabe's metric is the classical graph-theory cyclomatic number defined as:

v(G) = # edges - # nodes + 2 (# connected components). McCabe presents two simpler methods of calculating v(G): the number of predicate nodes plus 1 or the number of regions computed from a planar graph of the control flow.

Length. The length of the program was the total number of Fortran statements, excluding comments. The total number of executable statements was found to be highly correlated with number of statements ($r = 0.99$, $p \leq 0.001$).

Dependent Variable

The dependent variable was the number of minutes necessary for the participant to locate and correct the bug.

Analysis

The analysis of data was conducted in two phases. The first phase was an experimental test of the independent variables, while the second phase evaluated the software complexity metrics. In the first phase, experimental data were analyzed in a hierarchical regression analysis. In this analysis, domains of variables were entered sequentially into a multiple regression equation to determine if each successive domain significantly improved the predictive capability of the equation developed from domains already entered. Thus, the order in which domains were entered into the analysis was important. Variables representing the different conditions of experimentally manipulated variables were effect-coded (Kerlinger and Pedhazur, 1973).

The second phase of analysis investigated relationships between the time to find the bug and the metrics, Halstead's E, McCabe's v(G), and number of statements in the program. All correlations are Pearson product-moment correlations.

RESULTS

Preliminary Tasks

Group 1 (Participants 1-27) and Group 2 (Participants 28-54) were given different preliminary tasks. The two algorithms were of varying difficulty, producing significant differences in both time to completion and percent of completions. Finding the bug in the greatest common divisor algorithm required an average of 23.8 minutes with 22% failing to find the bug in 45 minutes, while the sorting algorithm required only 14.6 minutes with only 4% failing to find the bug. However, no significant differences in performance between the two groups occurred on the experimental programs.

Experimental Manipulations

The average time to locate bugs across all experimental conditions was 20.1 minutes (SD = 16.2). All but six of the 162 experimental tasks comprising this experiment were completed successfully during the allotted 60 minutes. These six conditions were not associated with any particular factor.

Despite the use of a preliminary task to familiarize the participants with the experiment, a significant order effect occurred ($p \leq 0.04$), indicating that learning took place during the first of the three experimental tasks (Fig. 5).

Results of a hierarchical regression analysis of the independent variables on the time to find the bug are presented in Table 1. Differences in solution time for the three programs were significant ($p \leq 0.01$). Finding the bug in the accounting program required an average of 15.1 minutes, 20.0 minutes in the program that sorted questionnaire data, and 25.0 minutes in the grade-scoring program. Increasing the length of the programs had a modest effect ($p \leq 0.06$) on the time to locate and correct the error. The average time for the short program was 16 minutes, while the medium and long programs required a mean of 21 and 23 minutes, respectively.

Averages for the three error categories were not significantly different from one another. However, a very large interaction occurred between type of bug and program (Fig. 6). This interaction accounted for the largest percent of variance (26%) of any of the experimental relationships studied. No significant differences in performance resulted from the three types of control flow.

Software Complexity Metrics

Intercorrelations among the three measures of software complexity were computed from the 27 different versions of the programs at both the subroutine and program levels (Table 2). Substantial intercorrelations were observed among Halstead's E, McCabe's v(G), and length at the subroutine level. When computed on the total program, the correlation between length and McCabe's v(G) increased, while the correlations for Halstead's E with these two measures were substantially smaller, especially with lines of code.

Correlations between time to find the bug and the complexity metrics were calculated for unaggregated data (three experimental tasks for each of the 54 participants, $n = 162$) and for data averaged over the six scores obtained for each program (Table 3). Correlations for the aggregated data were much higher than those for the unaggregated scores. All three metrics predicted performance equally well at the subroutine level. At the program level, however, E was the best predictor, accounting for more than twice the variance in performance than did the length (56% versus 27%, respectively). The variance accounted for by v(G) fell between these values (42%). A stepwise multiple regression analysis indicated that length and v(G) added no increments to the prediction afforded by E.

The scatterplot of performance with Halstead's E presented in Figure 7 suggested the existence of a curvilinear trend in the data. The significance of this trend was tested using the second degree polynomial regression approach suggested by both Cohen and Cohen (1975) and Kerlinger and Pedhazer (1973) for investigating curvilinear relationships. A multiple correlation coefficient of 0.84 indicated that the curvilinear trend accounted for an additional 15% ($p \leq 0.001$) of the

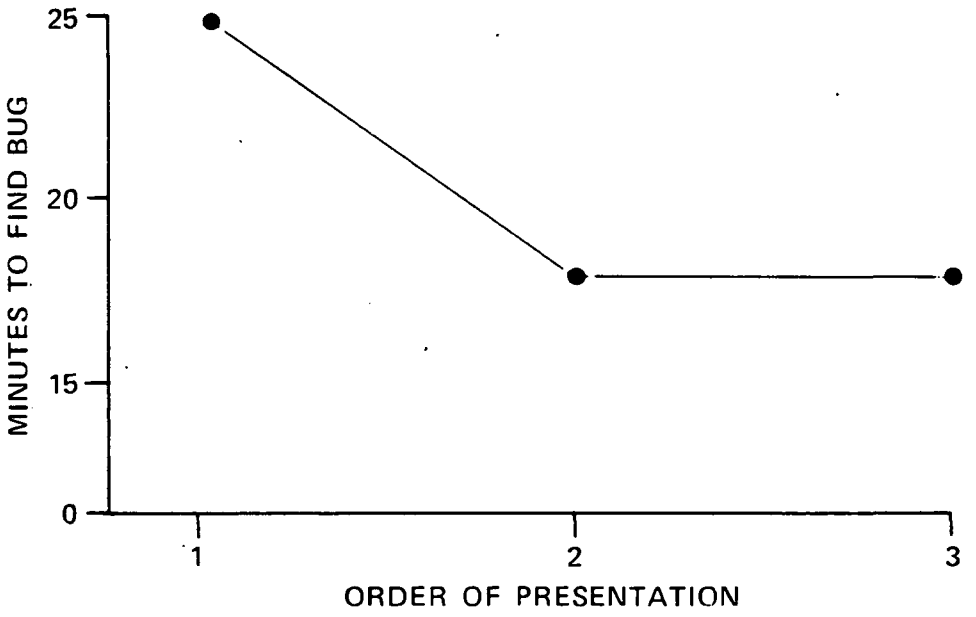


Figure 5. Order Effect on the Three Experimental Tasks

Table 1

Hierarchical Regression Analysis for Time to Find Bug

Variable	df	R^2	ΔR^2
(1) Program	2	0.06**	0.06**
(2) Presentation order	2	0.04*	0.04*
(3) Type of bug	2	0.00	0.00
(4) Program X bug interaction	4	0.26***	0.26***
(5) Complexity of control flow	2	0.02	0.02
All variables	12		0.38***

NOTE: $n = 162$. R^2 column represents the separate regression for each domain.

*$p \leqslant 0.05$
**$p \leqslant 0.01$
***$p \leqslant 0.001$

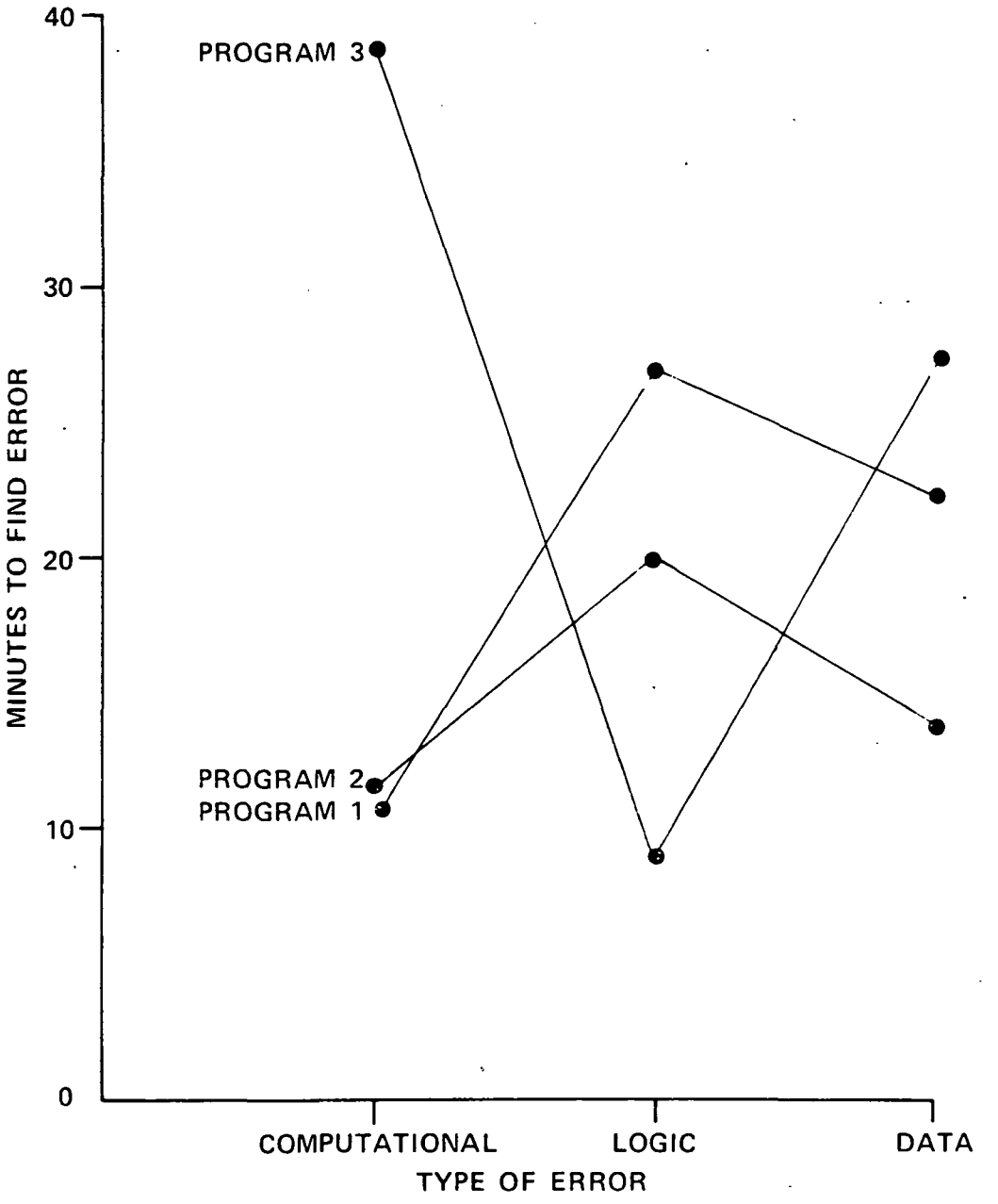


Figure 6. Program by Error Interaction

Table 2

Intercorrelations Among Complexity Metrics

Metrics	Correlations	
	E	v(G)
Subroutine:		
v(G)	0.92***	
Length	0.89***	0.81***
Program:		
v(G)	0.76***	
Length	0.56***	0.90***

NOTE: n = 27.

***p ≤ 0.001

Table 3

Correlation Between Performance Time and Complexity Metrics

Metric	Correlations	
	Unaggregated (n = 162)	Aggregated (n = 27)
Subroutine:		
Halstead's E	0.25***	0.66***
McCabe's v(G)	0.24***	0.63***
Length	0.25***	0.67***
Program:		
Halstead's E	0.28***	0.75***
McCabe's v(G)	0.25***	0.65***
Length	0.20**	0.52**

**p ≤ 0.01
***p ≤ 0.001

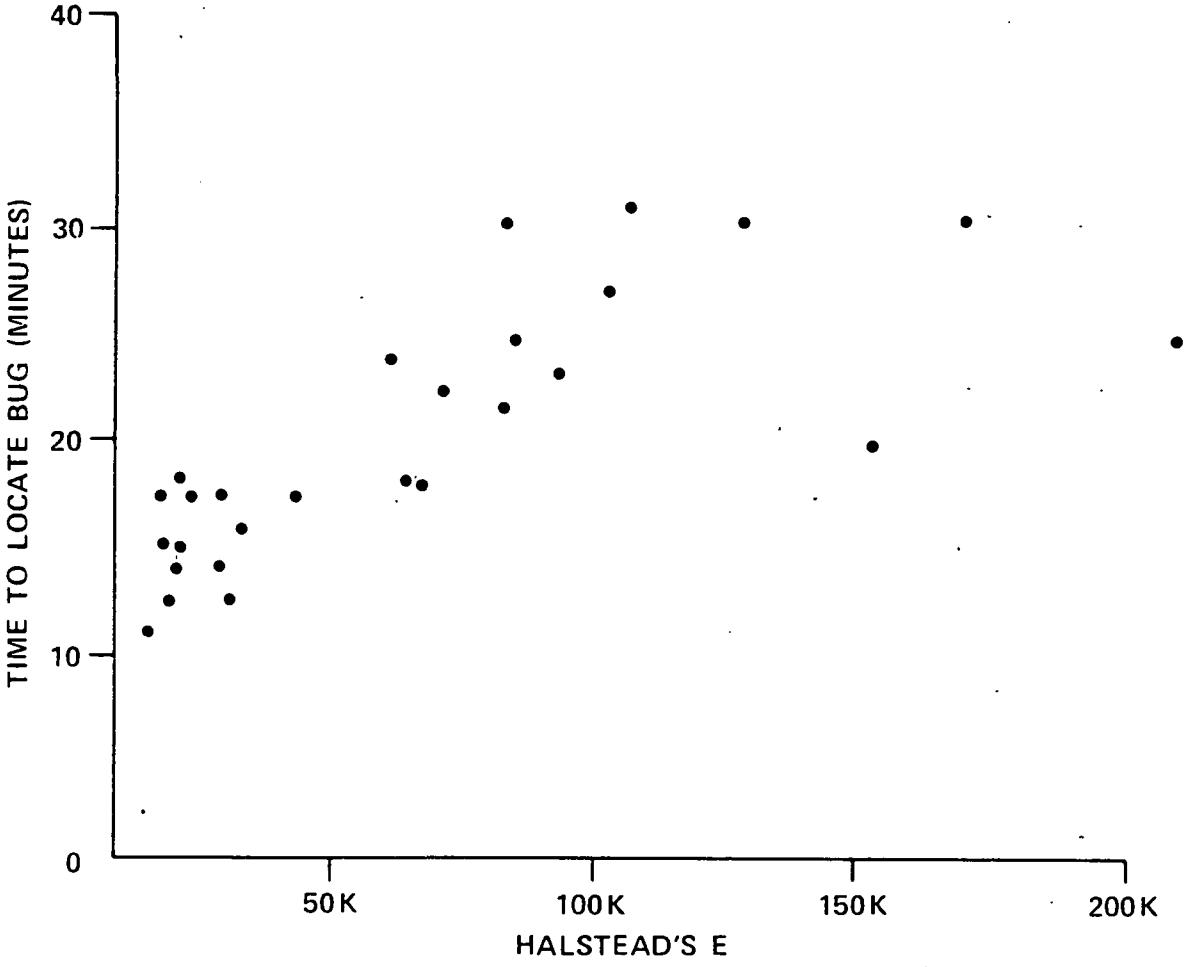


Figure 7. Scatterplot of Halstead's E and Performance

variance beyond that accounted for by a linear relationship. The prediction equation generated from these data was:

$$\text{minutes to find bug} = 9.837 + 0.00239\underline{E} - 0.00000000079\underline{E}^2$$

However, with few data points in the right tail of this distribution for Halstead's $\underline{E}$, it is difficult to extrapolate to the exact shape of the curvilinear trend. No curvilinear trend was detected with either the lines of code or McCabe's $\underline{v(G)}$.

Experiential Factors

The relationship between complexity metrics and performance was investigated within groups of programmers differing in years of professional experience programming in Fortran. As a heuristic, the participants were divided into two groups of approximately equal numbers: those with three or fewer years experience and those with more than three years experience. The results presented in Table 4 indicate that the complexity measures were more predictive of performance for less experienced programmers, especially when computed at the subroutine level.

Two measures of experience were also found to be related to the performance of less experienced programmers (Table 5), but not to the performance of experienced programmers. The first such measure was the number of programming languages the participant knew. The second metric was the number of items checked on the experience questionnaire. The moderating effects of programmer experience may have been the result of greater variability in performance for programmers with less experience (Fig. 8). This greater variability would increase the ability of correlational tests to detect significant relationships (Cohen and Cohen, 1975).

DISCUSSION

Four factors were found to influence the speed with which programmers could find a bug in a computer program. These factors were order of presentation, specific program, a program by error interaction, and the complexity of the code as measured by software complexity metrics. Type of bug and type of control flow, however, did not account for a significant proportion of the variation in performance.

Variance in programmer performance associated with differences among the programs replicated results from two previous experiments in this series (Sheppard, et al., 1979). However, a much larger percent of the variance in performance was accounted for by a program by error interaction. It appeared that some quality of the algorithm in which the bug was embedded influenced a programmer's ability to locate it. The time required to detect similar errors contained in similar statements depended on the program in which the error was embedded. This result has implications for the usefulness of various schemes for categorizing software bugs. The implied value of these taxonomies is to identify properties of bugs which suggest how they are created or how difficult they are to detect. Simple taxonomies based on syntactic relationships will probably not prove sufficient for this purpose. The results of this experiment suggest that the detectability of a bug depends on the context of the algorithm surrounding it. This contextual effect may determine the optimal search strategy for finding the bug, and it is this search strategy that needs to be understood if debugging performance is to be improved.

Table 4

Correlations Between Performance and Complexity Metrics Moderated by Years of Fortran Experience

Metrics	Correlations	
	⩽3 Years (n = 75)	>3 Years (n = 87)
Subroutines:		
Halstead's E	0.39***	0.11
McCabe's v(G)	0.37***	0.07
Length	0.33***	0.17
Program:		
Halstead's E	0.38***	0.20*
McCabe's v(G)	0.29***	0.21*
Length	0.18	0.22*

NOTE: Dividing the data into groups of programmers required that scores be analyzed on individual tasks rather than on tasks averaged by program. Thus, this analysis was performed on the 75 experimental tasks performed by the 25 participants with 3 or fewer years of Fortran experience and the 87 tasks performed by the 29 participants with more than 3 years experience.

*$p \leqslant 0.05$
**$p \leqslant 0.01$
***$p \leqslant 0.001$

Table 5

Relationships of Experiential Factors to Performance for Programmers Differing in Fortran Experience

Relevant Experience	⩽3 Years (n = 25)	>3 Years (n = 29)	Total (n = 54)
# of Programming Languages	-0.49**	-0.03	-0.19
Questionnaire Score	-0.48**	-0.11	-0.33**

**$p \leqslant 0.01$

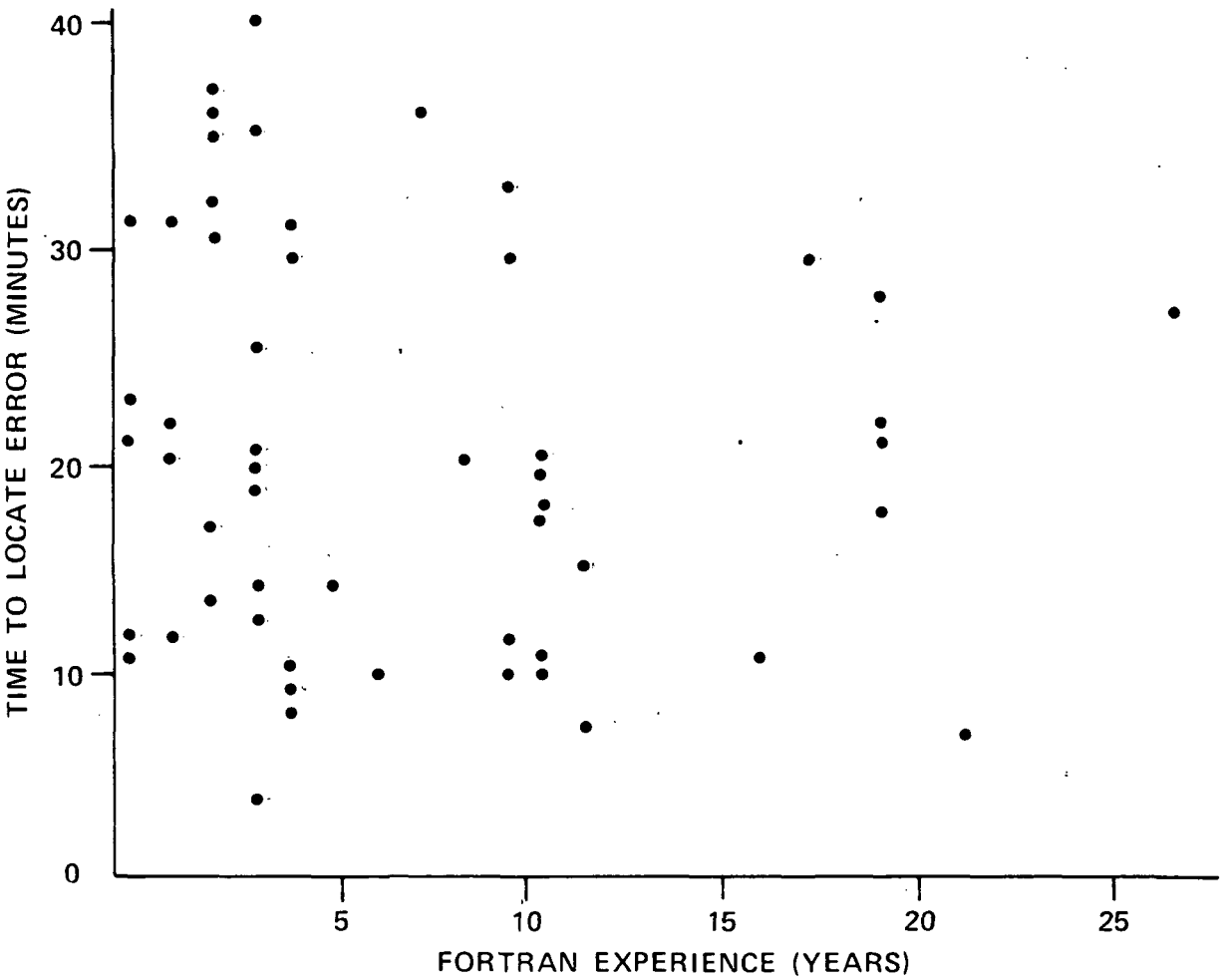


Figure 8. Scatterplot of Experience and Performance

In the last section of the post-session questionnaire, the participants were asked to describe their searching strategies for locating the bugs. Typically, one of two approaches was described. In the first strategy the programmer tried to understand the whole program from beginning to end before searching for the section with the bug. In the second strategy the programmer used appropriate clues in the output to go directly to the section containing the bug. The latter appeared to be a much quicker strategy for debugging, but there were insufficient data for a meaningful statistical analysis. In order to improve the debugging performance of programmers it will be important not only to identify effective search strategies, but also to identify conditions under which they will be differentially effective.

No significant differences were evident among the three types of top-down control flow tested in this experiment. This finding agrees with previous results (Sheppard, et al., 1979) where differences were found between top-down and convoluted control flow, but not between types of top-down control flow. The minor deviations from strictly structured coding allowed in the naturally structured version of this experiment did not adversely affect performance. Summarizing the combined results of the three experiments, it would appear that the overall top-down quality of the control flow is important to performance, but careful attention to strict structuring does not appear to improve programmer performance significantly.

Since no difference was found between the graph-structured and Fortran 77 program versions, it would appear that the newer constructs provide little additional aid in a debugging task beyond that provided by a top-down flow. Only five of the 54 participants had previously used Fortran 77, so a lack of familiarity with the new constructs may have prevented them from finding the bug more quickly in Fortran 77 than in Fortran IV. However, immediately prior to the experiment a short training session was conducted with each group of participants in which the new Fortran 77 constructs were discussed in detail. These constructs were similar to those implemented in Fortran IV, and the participants' previous lack of familiarity with them was probably not a significant factor in their performance.

Most laboratory studies exhibit a certain degree of artificiality that is necessary for experimental control. In this experiment participants were told there was only one bug in a program. While this situation differs from a normal programming environment, it should not have affected participant's ability to perform the tasks. These experimental tasks may have been simpler to perform than typical debugging problems since there was greater certainty about the bugs. Further, differences between the correct and erroneous output were clearly marked on the erroneous output, reducing the amount of comparison necessary to discover what problems had occurred.

During a typical debugging problem a programmer could refer to the functional specifications for a program or to comments included in the code. However, no such aids were made available in this experiment. The participant's comprehension of the program's function had to be gleaned from the code or from the input and output listings. The latter were designed to be self-explanatory, with each section labeled appropriately; e.g., "FINAL COURSE GRADE" or "TRIAL BALANCE." Although adding some artificiality to the experimental situation, the absence of documentation was an attempt to equalize the amount of information provided by materials other than the code.

Software Complexity Metrics

The results of this experiment not only replicated the results obtained in our previous research, but also demonstrated that more viable results could be obtained when limitations in our earlier

experimental procedures were overcome. For instance, our previous research was conducted exclusively on small-sized (35–55 lines of code) programs, which seems to have limited the results in three ways. First, the range of values on the factors studied in those programs seems to have been too restricted to detect the size of relationships observed here. Second, the curvilinear relationship observed in this experiment between Halstead's E and performance would not have been observed if longer programs had not been used in the experimental tasks. Third, the extremely high intercorrelation between length and Halstead's E at the subroutine level suggests that both are measuring program volume. With larger programs the information measured appears to differ; that is, Halstead's E measures something in addition to, but inclusive of, factors measured by length.

Many small-sized programs can be grasped by the typical programmer as a cognitive gestalt. The psychological complexity of such programs is adequately represented by the volume of the program in terms of the number of lines of code. When the code grows beyond a subroutine, its complexity to the programmer is better assessed by measuring constructs other than the number of lines of code. This may result partly because programmers cannot grasp the entire program within their mental spans at a single time. For larger programs the difficulty programmers experience is better represented by counts of operators, operands, and control paths. Thus, as the size of a program increases, Halstead's E seems to be a better measure of its psychological complexity.

One possible explanation for the superior predictive ability of Halstead's E is that the relationship between program size and performance is curvilinear, and the algorithmic transformation with the Halstead measure captures this relationship while lines of code does not. There was no evidence in these data of a curvilinear relationship between lines of code and performance. On the other hand, a curvilinear relationship did exist between Halstead's E and performance. This trend suggests that as Halstead's E grows larger, a program becomes more psychologically complex, but the increments in difficulty grow smaller and smaller. In the experimental task used in this debugging experiment, there seemed to be an amount of time that was typically required to locate a bug within a subroutine once the correct subroutine had been identified (approximately 16 minutes). Added to this baseline rate was the time required to identify the proper subroutine. The curvilinearity of the relationship between time to find the bug and Halstead's E appeared to result from the time required to isolate the problem subroutine.

The moderating effects of experiential factors also replicated the results found in the earlier experiments. The metrics again proved to be better predictors of performance for programmers with three or fewer years experience in Fortran than for those with more than three years experience. It was also possible to predict the performance of an individual programmer from job history data. Several important factors seemed to be the number of languages a programmer had used and familiarity with certain programming concepts. These predictions from job history were also more valid for programmers who had three or fewer years of experience in Fortran. Future work is needed to refine the use of experiential questionnaires for use in personnel functions such as selection, assessment for training needs, and placement.

Code which is more psychologically complex may also be more error-prone and difficult to test. The results of this experiment provide evidence that the software complexity metrics developed by Halstead and McCabe are related to the difficulty programmers experience in locating errors in code. Thus these metrics appear to be capable of satisfying several practical applications. They can be used in providing feedback both to programmers about the complexity of the code they

have developed and to managers about the resources that will be necessary to maintain particular sections of code. Further evaluative research needs to assess the validity of these uses in ongoing software projects.

ACKNOWLEDGEMENTS

The authors are grateful to Judy McWilliams and Mary Anne Borst who helped with this experiment and to Beverly Day for manuscript preparation. We are also grateful to Dr. Gerald Hahn for advice on experimental design, to Drs. Tom Love and Ben Shneiderman for advice on the experimental tasks and procedures, and to Dr. John O'Hare for his careful review of this report. We are especially appreciative of the efforts of Earl North and Leo Pompliano of General Electric; Jan Gombert of Applied Urbanetics; Mrs. Joan Shields, Cols. William Eglington, Earl Goetze and Richard Blair, and Lt. Col. Pat Harris of the U.S. Air Force; and Capt. Webster and J. Rehbehn of the U.S. Navy in providing the participants for this research. The support and encouragement of both Gerald Dwyer and Lou Oliver has been vital to the success of this research.

This research was supported by the Office of Naval Research, Engineering Psychology Programs (Contract #N0014-77-C-0158). The views expressed in this paper, however, are not necessarily those of the Office of Naval Research or the Department of Defense.

REFERENCES

Brainerd, W., Fortran 77. Communications of the ACM. 1978, 21, 806-820.

Brooks, R. Unpublished algorithm. Irvine, CA: University of California at Irvine, Computer Science Department, 1978.

Campbell, D. and J. C. Stanely, Experimental and quasi-experimental designs for research. Chicago: Rand-McNally, 1967.

Carlson, W. E. and B. DeRoze, Defense system software research and development plan. Unpublished manuscript, Arlington, VA: Defnese Advanced Research Projects Agency, September 1977.

Cohen, J. and P. Cohen, Applied multiple regression/correlation analysis for the behavioral sciences. New York: Wiley, 1975.

Curtis, B., S. B. Sheppard, P. Milliman, M. A. Borst, and T. Love, Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics. IEEE Transactions on Software Engineering, 1979, 5, 95-104.

Department of Defense requirements for high order computer programming languages: Revised "IRONMAN." SIGPLAN Notices, 1977, 12, 39-54.

DeRoze, B., Software research and development technology in the Department of Defense. Paper presented at the AIIE Conference on Software, Washington, D.C.: December 1977.

Dijkstra, E. W., Notes on structured programming. In Structured programming, O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, (Ed.) New York: Academic, 1972.

Fitzsimmons, A. B. and L. T. Love, A review and evaluation of software science. ACM Computing Survey, 1978, 10, 3-18.

Gordon, R. D., A measure of mental effort related to program clarity. Unpublished doctoral dissertation, Purdue University, 1977.

Gould, J. D., Some psychological evidence on how people debug computer programs. International Journal of Man-Machine Studies, 1975, 7, 151-182.

Gould, J. D. and P. Drongowski, An exploratory study of computer program debugging. Human Factors, 1974, 16, 258-277.

Halstead, M. H., Elements of software science. New York: Elseiver North-Holland, 1977.

Hecht, H., W. A. Sturm, and S. Trattner, Reliability measurement during software development. Redondo Beach, CA: Aerospace Corp., 1978.

Kerlinger, F. N. and E. J. Pedhazur, Multiple regression in behavioral research. New York: Holt, Rinehart, and Winston, 1973.

McCabe, T. J., A complexity measure. IEEE Transactions on Software Engineering, 1976, 2, 308-320.

Nolen, R. L., Fortran IV computing and applications. Reading, MA: Addison-Wesley, 1971.

Ottenstein, K. J., A program to count operators and operands for ANSI-FORTRAN modules (Tech. Rep. CSD-TR-196). West Lafayette, IN: Purdue University, Computer Science Department, 1976.

Sheppard, S. B., B. Curtis, M. A. Borst, P. Milliman, and L. T. Love, First year results from a research program on human factors in software engineering. In Proceedings of the 1979 National Computer Conference, Montvale, NJ: AFIPS, 1979.

Sheppard, S. B. and L. T. Love, A preliminary experiment to test influences on human understanding of software. In Proceedings of the 21st Meeting of the Human Factors Society. Santa Monica, CA: Human Factors Society, 1977.

Tenny, T., Structured programming in FORTRAN. Datamation, 1974, 20, 110-115.

The military software market (Rep. 427). New York: Frost and Sullivan, 1977.

Veldman, D. J., Fortran programming for the behavioral sciences. New York: Holt, Rinehart, and Winston, 1967.

Wescourt, K. T. and L. Hemphill, Representing and teaching knowledge for troubleshooting/ debugging (Tech. Rep. 292). Stanford, CA: Stanford University, Institute for Mathematical Studies in Social Science, 1978.

Youngs, E. A., Human errors in programming. International Journal of Man-Machine Studies, 1974, 6, 361-376.

APPENDIX A

PRETEST

Sorting Algorithm

INPUT

DATAPRE

```
 25
110
 30
 31
  1
153
193
 62
 78
 16
  1
193
 62
 78
 74
168
192
199
999
  5
 78
 79
 56
  9
 57
  3
```

```
100         IMPLICIT INTEGER(A-Z)
110         DIMENSION A(50),B(50)
115         READ('DATAPRE',10) N
116         DO 5 I = 1, N
120     5   READ('DATAPRE',10) A(I)
130    10   FORMAT(I3)
140         DO 100 J = 1, N
160         SMALL = A(1)
170         M = 1
180         DO 20 K = 2,N
190    15   IF(A(K) .LT. SMALL) GO TO 20
200         SMALL = A(K)
210         M = K
220    20   CONTINUE
230         B(J) = SMALL
240         A(M) = 1000
250   100   CONTINUE
251         DO 101 I = 1, N
260   101   PRINT 110, B(I)
261   110   FORMAT(2X,I4)
270         STOP
280         END
```

CORRECT OUTPUT	INCORRECT OUTPUT
1	999
1	1000
3	1000
5	1000
9	1000
16	1000
30	1000
31	1000
56	1000
57	1000
62	1000
62	1000
74	1000
78	1000
78	1000
78	1000
79	1000
110	1000
153	1000
168	1000
192	1000
193	1000
193	1000
199	1000
999	1000

S. Sheppard
G.E.

APPENDIX A

PRETEST

INTRODUCTION

DDI is a software development company currently in charge of maintaining the Advanced Orbit Ephemeris Subsystem (AOES) for the USAF. This presentation will address the various methods used in maintaining and upgrading the AOES, and to show how these methods reduce the number of discrepancies in the AOES.

MAINTAINING THE EXISTING SYSTEM

Requirements are generated by the user community to either modify or upgrade the current AOES. These requirements can modify existing programs or create programs which are then added to the AOES. The development of these requirements into software programs are delivered to the Air Force on a scheduled date and this delivery is called a MODEL. Any discrepancies found in the current or past models are corrected using machine code. The machine code is later converted to a HOL (JOVIAL) for incorporation into a future model.

PROBLEMS FACED IN MAINTENANCE

The original programs were all written in a very non-structured manner. The program logic in most of the original programs are very difficult to follow since more than one programmer was involved in the original coding and subsequent modifications. The comments are obscure, non-meaningful or absent in some instances. Furthermore, many discrepancies are corrected without any thought to future modifications in the area of the fix or to the readability of the correction (i.e., the correction seems to appear out of place in the area in which it occurs).

It mentioned in the introduction the objective of DDI was to try to alleviate discrepancies against delivered models. To do this Structured Software Techniques, and the formation of a Quality Assurance staff was implemented.

This presentation will describe these tools and their effectiveness and their weakness as they have been observed.

STRUCTURED SOFTWARE TECHNIQUES

- CURRENT METHODS
  - Top Down Design
    - Software Engineering, group leader and programmer sit down and review the requirement(s) for a new software program or for modifications to existing software programs. All major areas of the requirements are identified, and

these are further subdivided into lesser tasks. This process is repeated until each task can be dealt with separately.

- Effectiveness
  - Early in the development cycle all major interfaces for the requirements can be identified.
  - Any design trade–offs will surface and can be further analyzed.
  - Having the programmers involved gives them a better understanding of the possible problem areas and greater involvement in the final delivered product.

- STRUCTURED PROGRAMMING TECHNIQUES
  - The higher order language that is used by DDI is called JOVIAL. This language is very readable, flexible and very well suited for structured programming. The only constructs missing to truly make it an ideal structured langugae are the DO WHILE and CASE instructions.
  - The following guidelines are followed in the modification or development of software.
    - Comments
      - Have meaningful comments.
      - A comment should appear in at least every 3 or 4 lines of JOVIAL code, for every conditional statement and block structure.
      - Comment all items, arrays and tables.
    - All Items, Variables, Arrays and Tables
      - Alphabetized within their respective groups.
      - Distinct and have meaningful names.
      - Start them all in Column 4.
      - Overlays and defines are to be at the end of the parameter list.
      - Table entries should follow the indentation rules. Also, the presets of of tables and arrays.

- JOVIAL Executable Statements
  - Start in Column 4.
  - Are assigned to one line, and if more than one line is required, indent the continuation line at least 3 columns.
  - GOTO statements should be used with discretion.
- Conditional Statements and Block Structures
  - Indent statements following conditional statements by a minimum of 3 columns.
  - Indent block structure by 3 columns and identify the begin and end of each block.
- PROCS and CLOSES (Internal subroutines)
  - Whenever there is a choice use PROCS.
  - Start the Statement PROC or CLOSE in Column 1.
  - Whenever feasible try to pass single input and single output parameter.
  - Do not use the same input and output names in several PROCS.
  - Attempt to alphabetize your PROCS at the end of your program.
  - JOVIAL code, ITEMS, tables and arrays should start in Column 4.
  - For each PROC or CLOSE describe its purpose and all of its input and output parameters.
- Statement Labels
  - Start in Column 1.
  - Be assigned an individual line.
  - Have descriptive names.
  - For those in PROCs or CLOSEs the first few characters of the name can be used within that PROC or CLOSE.

- Effectiveness
  - Typographically, the program is more readable.
  - Programs are more readily understood.
  - Debugging and maintenance is greatly simplified.
  - Modifications can be more easily performed.

- STRUCTURED WALK-THROUGHS
  - After the programmer has coded his program a walk through of the code is performed between the programmer and the respective group leader.
  - *After the first clean compilation another program walk-through is exercised.*
  - During the final check-out phase a final walk-through is performed.
  - A walk-through of the developmental test deck is also performed to insure that the programmer test methods do indeed test those requirements and their interfaces of the program.
  - Effectiveness
    - To insure that the programmer has coded to meet the requirements.
    - Provide a check to determine if structured software guidelines are being performed.
    - Final walk-through is an insurance step to determine if any code change has affected meeting software requirements.
    - Development test deck walk-throughs insure more discrete or better testing methods by collapsing or expanding certain tests, or by adding new tests.

- PROGRAMMER NOTEBOOK
  - This is a text of information given to, created by or used by the programmer in developing programs for a development cycle. The contents include:
    - Schedules
    - Requirements
    - All design modifications
    - Initial data flow

- All documentation and their review comments and responses
- Any conversations concerning their program with outside agencies
- Data of program walk-throughs.

- Effectiveness
  - The programmer, group leader, software engineer or project director can assess materials used in the development of each program.
  - Historical records provide insight into an individual's thoughts and logic.
  - Programmers can refer to the notebook for insight for future modifications to the same program.
  - Especially useful if an individual leaves in the middle of the development and another individual must finish the development.

- TOP DOWN TESTING
  - This is the method of testing of all top level program modules before lower level modules are tested. Top down testing allows the testing of major interfaces first. Coding for a program need not be complete before top down testing can start, since stubs can be used.
  - Not all testing is done in a top down manner, in particular instances where a lower module performs some critical processing that is required at the upper levels, those lower programs are tested first using a driver program. But once those lower level programs have been tested, top down testing resumes.
  - Effectiveness
    - Both coding and testing can occur at the same time, and this leads to a better distribution of testing time.
    - Eliminates the need for driver programs to be written in order to check out the actual program.

METHODS TRIED BUT NON-EFFECTIVE FOR OUR WORK

Pseudo Code or Program Design Language

JOVIAL language can be used as a program design language and many programmers were getting too detail oriented and not looking at the structure of the program.

Flow Charting

Again, flow charting made the programmers detail oriented and not structure oriented. Flow block diagrams were only major blocks and decisions proved to be much more effective.

SOFTWARE QUALITY ASSURANCE GROUP

A software quality assurance (QA) group was created to formally validate the requirements of a model. The QA staff is a separate group of individuals whose task is to support the software development of the model. This is achieved by having a member of the QA staff sit in when the top-down design of a program is being done. This will aid the QA member to understand the requirements of the program. This understanding will be used in developing a formal system level validation test of the requirements. The QA staff will be responsible to execute all of their validation tests to verify that the user requirements have been satisfied. The QA staff has the responsibility to review all formal documentation produced by the programmers to insure that all requirements have been addressed and that the document conforms to the proper format. The QA group will be the configuration control point for each model.

Effectiveness

- Formal validation of the software requirements are centralized in a single document.
- Independent testing of software programs before a formal release.
- All discrepancies found can be more easily duplicated and solved by the programmers using a HOL.
- Configuration management control.

SUMMARY

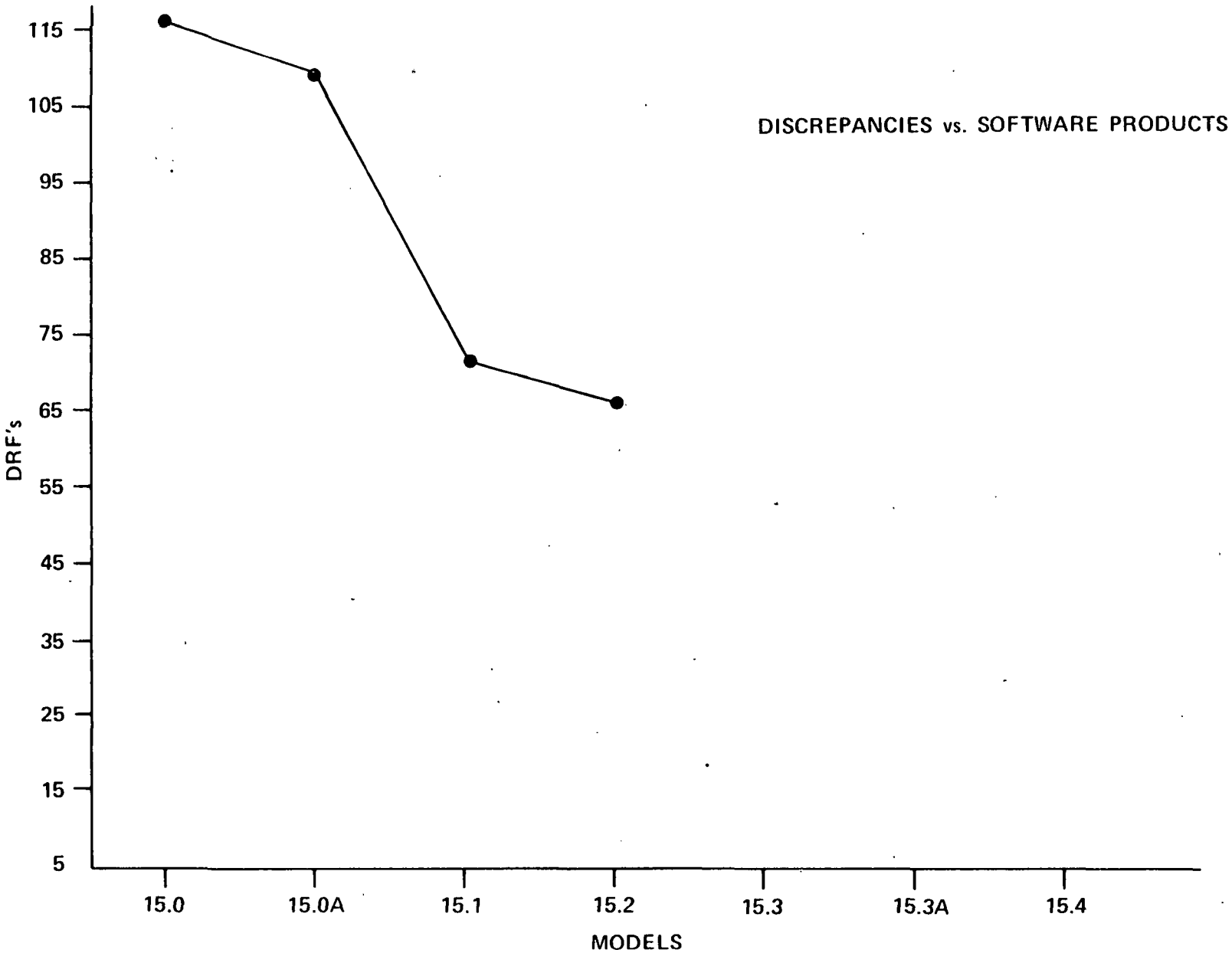
Using certain structured techniques with the added independent testing performed by the QA staff, DDI has reduced the number of discrepancies in modifying or upgrading our current system. There is a very definite advantage to applying these techniques to existing systems.

- MAINTAINING THE EXISTING SYSTEM
  - MODIFICATION TO EXISTING SOFTWARE
  - DEVELOPMENT OF NEW SOFTWARE TO AUGMENT THE CURRENT SYSTEM
  - CORRECT DESCREPANCIES FOUND IN THE CURRENT SYSTEM
- PROBLEMS FACED IN MAINTENANCE
  - ALL ORIGINAL PROGRAMS WRITTEN WITHOUT STRUCTURED TECHNIQUES
  - PROGRAM LOGIC IS DIFFICULT TO FOLLOW
  - OBSCURE COMMENTS OR NO COMMENTS
  - PATCHED AREAS
- STRUCTURED SOFTWARE TECHNIQUES
  - TOP DOWN DESIGN
    - MAJOR AREAS ARE IDENTIFIED
    - EFFECTIVENESS
      - OVERVIEW OF THE PROGRAM STRUCTURE
      - INTERFACES CAN BE IDENTIFIED EARLY
      - DESIGN TRADE-OFFS SURFACE
      - EARLY PROGRAMMER INVOLVEMENT
    - DRAWBACK
      - TOO MUCH MODULARIZATION

- STRUCTURED SOFTWARE TECHNIQUES
  - STRUCTURED PROGRAMMING TECHNIQUES
    - HOL - JOVIAL
      - COMMENTS ARE TO BE MEANINGFUL AND PLENTIFUL
      - INDENTATION OF CODE FOR CONDITIONAL STATEMENTS AND BLOCK STRUCTURES
      - MEANINGFUL NAMES FOR STATEMENT LABELS, INTERNAL SUBROUTINES, AND VARIABLES
    - EFFECTIVENESS
      - READABLE PROGRAMS
      - PROGRAM LOGIC MORE READILY UNDERSTOOD
      - DEBUGGING AND MAINTENANCE SIMPLIFIED
      - MODIFICATIONS MORE EASILY PERFORMED
    - DRAWBACKS
      - SYSTEM AND CORE LIMITATION
      - TIMING REQUIREMENTS

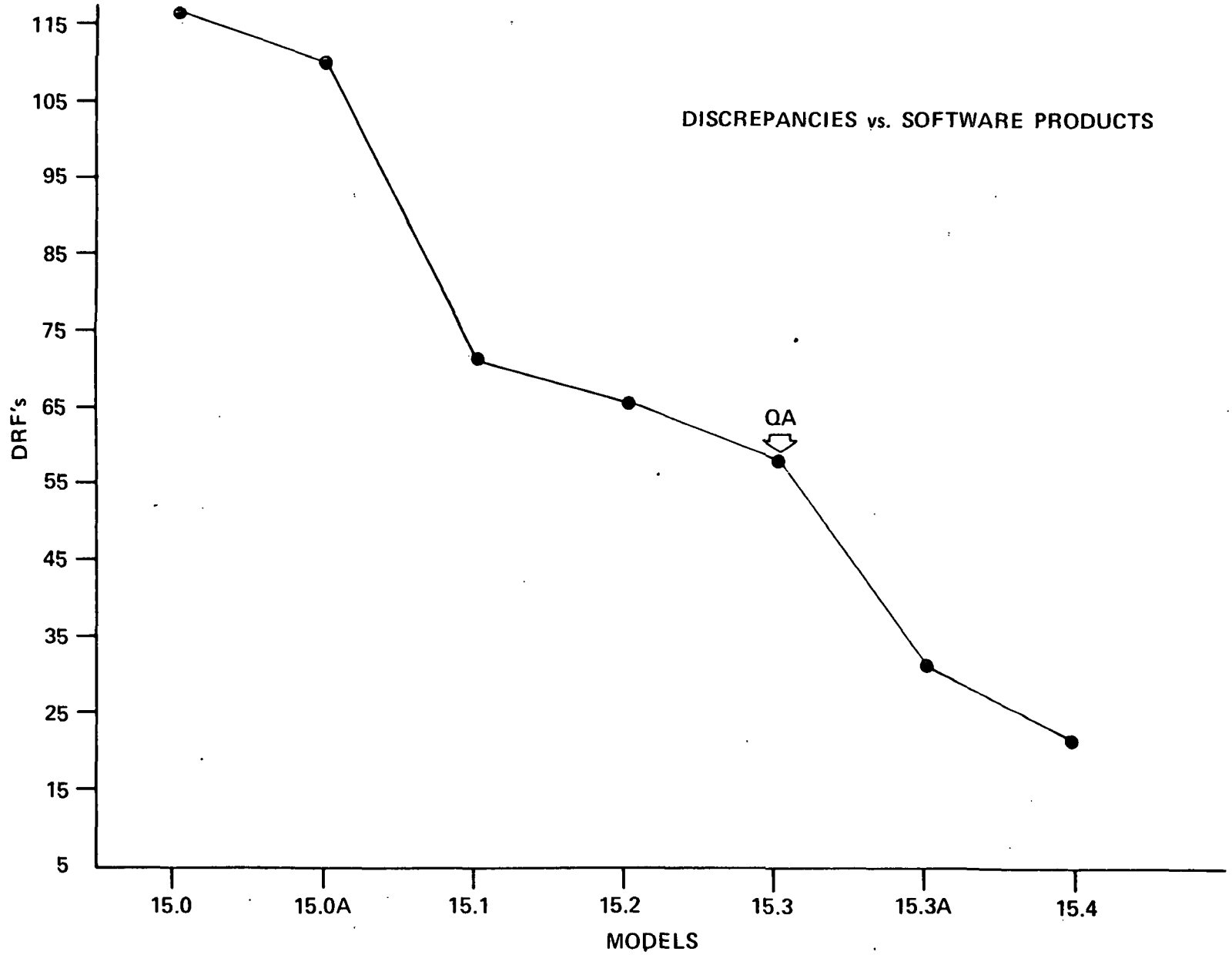
- STRUCTURED SOFTWARE TECHNIQUES
  - STRUCTURED WALK-THROUGHS
    - PROGRAM WALK-THROUGHS
      - AT LEAST THREE TIMES
    - DEVELOPMENT TEST DECK WALK-THROUGH
    - EFFECTIVENESS
      - PROGRAMMER HAS CODE TO MEET REQUIREMENTS
      - TESTING OF CODE WHICH SATISFY REQUIREMENTS
  - PROGRAMMER NOTEBOOK
    - TEXT OF INFORMATION USED TO SATISFY REQUIREMENTS
    - EFFECTIVENESS
      - HISTORICAL ACCOUNT OF PROGRAM DEVELOPMENT
      - USEFUL FOR SUBSEQUENT WORK ON THE SAME PROGRAM
      - USEFUL IF PROGRAMMER LEAVES BEFORE COMPLETION
    - DRAWBACK
      - PROGRAMMERS DO NOT ALWAYS UPDATE

- STRUCTURED SOFTWARE TECHNIQUES
  - TOP DOWN TESTING
    - USE TOP LEVEL MODULES TO TEST LOWER LEVEL MODULES
  - EFFECTIVENESS
    - MAJOR INTERFACES ARE TESTED FIRST
    - CODING DOES NOT HAVE TO BE COMPLETE, USE OF STUBS
    - ELIMINATION OF DRIVER PROGRAMS
    - BETTER DISTRIBUTION OF TESTING TIME
  - DRAWBACK
    - NOT ALL TESTING CAN BE DONE TOP DOWN


DISCREPANCIES vs. SOFTWARE PRODUCTS
DRF's
115
105
95
85
75
65
55
45
35
25
15
5
15.0
15.0A
15.1
15.2
15.3
15.3A
15.4
MODELS

- SOFTWARE QUALITY ASSURANCE (QA)
  - SUPPORT SOFTWARE DEVELOPMENT
    - UNDERSTAND REQUIREMENTS
    - FORMAL VALIDATION OF SOFTWARE REQUIREMENTS USING SYSTEM LEVEL TESTING
    - REVIEW DOCUMENTATION
    - CONFIGURATION CONTROL
  - EFFECTIVENESS
    - FORMAL TESTING IS CENTRALIZED
    - INDEPENDENT TEST
    - MINIMIZE DELIVERY PROBLEMS

W. Fujii
DDI


DISCREPANCIES vs. SOFTWARE PRODUCTS
DRF's
115
105
95
85
75
65
55
45
35
25
15
5
QA
15.0
15.0A
15.1
15.2
15.3
15.3A
15.4
MODELS